Technical Report

IRI-TR-07-02

Exploiting single-cycle symmetries in continuous constraint problems

Vicente Ruiz de Angulo
Carme Torras

Institut de Robòtica i Informàtica Industrial

Abstract

Symmetries in discrete constraint satisfaction problems have been explored and exploited in the last years, but symmetries in continuous constraint problems have not received the same attention. Here we focus on permutations of the variables consisting of one single cycle. We propose a procedure that takes advantage of these symmetries by interacting with a continuous constraint solver without interfering with it. A key concept in this procedure are the classes of symmetric boxes formed by bisecting a n-dimensional cube at the same point in all dimensions at the same time. We analyze these classes and quantify them as a function of the cube dimensionality. Moreover, we propose a simple algorithm to generate the representatives of all these classes for any number of variables at very high rates. A problem example from the chemical field and a kinematics solver are used to show the performance of the approach in practice.

Institut de Robòtica i Informàtica Industrial (IRI)
Consejo Superior de Investigaciones Científicas (CSIC)
Universitat Politècnica de Catalunya (UPC)
Llorens i Artigas 4-6, 08028, Barcelona
Spain
Tel (fax): +34 93 401 5750 (5751)
http://www-iri.upc.es

Corresponding author:
Vicente Ruiz de Angulo
tel: +34 93 401 5752
ruiz@iri.upc.edu
http://www-iri.upc.es/people/ruiz

Contents

1 Introduction **1**

2 Symmetry in Continuous Constraint Problems **1**

3 Box symmetry **2**
3.1 Box symmetry classes obtained by bisecting a n-cube 3

4 Algorithm to exploit box symmetry **3**

5 An illustrative example **4**

6 Analysis of $\mathbb{SR}_n$: Counting the number of classes **5**

7 Generating $\mathbb{SR}_n$, the classes of symmetric boxes **8**

8 Conclusions **12**

1 Introduction

Symmetry exploitation in discrete constraint problems has received a great deal of attention lately [7, 4, 5, 11]. On the contrary, symmetries have been largely disregarded in continuous constraint solving, despite the important growth in both theory and applications that this field has recently experienced [13, 1, 6, 10].

Continuous (or numerical) constraint solving is often tackled using Branch-and-Prune algorithms [14], which iteratively locate solutions inside an initial domain box, by alternating box subdivision (branching) and box reduction (pruning) steps. Motivated by a molecular conformation problem, in this paper we deal with the most simple type of box symmetry, namely that in which domain variables (i.e., box dimensions) undergo a single-cycle permutation leaving the constraints invariant. This can be seen, thus, as a form of constraint symmetry in the terminology introduced in [3].

2 Symmetry in Continuous Constraint Problems

We are interested in solving the following general continuous Constraint Satisfaction Problem (continuous CSP): Find all points $\mathbf{x} = (x_1, \ldots, x_n)$ lying in an initial box of $\mathbb{R}^n$ satisfying the constraints $f_1(\mathbf{x}) \in C_1 \ , \ldots, f_m(\mathbf{x}) \in C_m$, where f_i is a function $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$, and C_i is an interval in $\mathbb{R}$.

The only particular feature that we require of a Continuous Constraint Solver (CCS) is that it has to work with an axis-aligned box in $\mathbb{R}^n$ as input. Also, we assume that the CCS returns solution boxes. Note that a CCS returning solution points is a limit case still contained in our framework.

We say that a function $s : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is a point symmetry of the problem if there exists an associated permutation $\sigma \in \Sigma_m$ such that $f_i(\mathbf{x}) = f_{\sigma(i)}(s(\mathbf{x}))$ and $C_i = C_{\sigma(i)}$, $\forall i = 1, \ldots, m$. We consider symmetry as a property that relates points that are equivalent as regards to a continuous CSP. Concretely, from the above definition one can conclude that $\mathbf{x}$ is a solution to the problem iff $s(\mathbf{x})$ is a solution to the problem. Let s and t be two symmetries of a continuous CSP with associated permutations σ_s and σ_t. It is easy to see that the composition of symmetries $s(t(\cdot))$ is also a symmetry with associated permutation $\sigma_s(\sigma_t(\cdot))$.

An interesting type of symmetries are permutations (bijective functions of a set onto itself) of the components of $\mathbf{x}$. Let D be a finite set. A cycle of length k is a permutation ψ such that there exist distinct elements $a_1, \ldots a_k \in D$ such that $\psi(a_i) = \psi(a_{(i+1) mod\ k})$ and $\psi(z) = z$ for any other element $z \in D$. Such a cycle is represented as $(a_1, \ldots a_k)$. Every permutation can be expressed as a composition of disjoint cycles (i.e, cycles without common elements), which is unique up to the order of the factors. Composition of cycles is represented as concatenation, as for example $(a_1, \ldots a_k)(b_1, \ldots b_l)$. In this paper we focus on a particular type of permutations, namely those constituted by a single cycle. In its simplest form[1], this is $s(x_1, x_2, \ldots x_n) = (x_{\theta(1)}, x_{\theta(2)}, \ldots x_{\theta(n)}) = (x_2, x_3 ... x_n, x_1)$, where $\theta(i) = (i + 1)\ mod\ n$.

[1] In general, the variables must be arranged in a suitable order before one can apply the circular shifting. Thus, the general form of a single-cycle symmetry is $s(\mathbf{x}) = h^{-1}(g(h(\mathbf{x})))$, where $h(x_1, \ldots x_n) = (x_{\phi(1)}, \ldots, x_{\phi(n)})$, $\phi \in \Sigma_n$ is a general permutation that orders the variables, and $g(x_1, \ldots, x_n) = (x_{\theta(1)}, \ldots x_{\theta(n)})$ is the circular shifting above. Thus, the cycle ψ defining the symmetry can be expressed as $\psi = \phi^{-1}(\theta(\phi(\cdot)))$. Since the reordering does not change substantially the presented concepts and algorithms, we have simplified notation in the paper by assuming that the order of the component variables is the appropriate one, i.e., that $\psi = \theta$.

Example: $n = 3, m = 4, \mathbf{x} = (x_1, x_2, x_3) \in [-1, 1] \times [-1, 1] \times [-1, 1]$,

$$\begin{aligned} f_1(\mathbf{x}): \quad & x_1 2 + x_2 2 + x_3 2 \in [5, 5] \equiv \; x_1 2 + x_2 2 + x_3 2 = 5 \\ f_2(\mathbf{x}): \quad & 2x_1 - x_2 \in [0, \infty] \equiv \; 2x_1 - x_2 \geqslant 0 \\ f_3(\mathbf{x}): \quad & 2x_2 - x_3 \in [0, \infty] \equiv \; 2x_2 - x_3 \geqslant 0 \\ f_4(\mathbf{x}): \quad & 2x_3 - x_1 \in [0, \infty] \equiv \; 2x_3 - x_1 \geqslant 0 \end{aligned}$$

There exists a symmetry $s(x_1, x_2, x_3) = (x_2, x_3, x_1)$, for which there is no need of reordering the variables. The constraint permutation associated to s is $\sigma(1) = 1$, $\sigma(2) = 3$, $\sigma(3) = 4$, and $\sigma(4) = 2$.

For $n \geqslant 3$ there is never a unique symmetry for a given problem. If there exists a symmetry s, then for example $s2(\mathbf{x}) = s(s(\mathbf{x}))$ is another symmetry. In general, using the convention of denoting $s0(\mathbf{x})$ the identity mapping, $\{s^i(\mathbf{x}), i = 0 \ldots n-1\}$ is the set of different symmetries that can be obtained composing $s(\mathbf{x})$ with itself, while for $i \geqslant n$ we have that $s^i(\mathbf{x}) = s^{i \bmod n}(\mathbf{x})$. Thus, a single-cycle symmetry generates by composition $n-1$ symmetries, excluding the trivial identity mapping. Some of them may have different numbers of cycles. Imagine for example that in a continuous CSP with $n = 4$ the permutation of variables (1 2 3 4) is a symmetry. Then, the permutation obtained by composing it twice, (1 3)(2 4), is also a symmetry of the problem, but has a different number of cycles, and the longest cycle has length two instead of four. Besides, the former permutation cannot be generated from the latter. The algorithm presented in this paper deals with all the compositions of a single-cycle symmetry, even if some of them are not single-cycle symmetries. The gain obtained with the proposed algorithm will be shown to be ideally proportional to the number of different compositions of the selected symmetry. Therefore, when several single-cycle symmetries exist in a continuous CSP problem, the algorithm should be used with that generating the most symmetries by composition, i.e., with that having the longest cycle.

3 Box symmetry

Since continuous constraint solvers work with boxes, we turn our attention now to the set of points symmetric to those belonging to a box $\mathcal{B} \subseteq \mathbb{R}^n$. [2]

Let s be a single-cycle symmetry corresponding to the circular variable shifting θ introduced in the preceding section, and $\mathcal{B} = [\underline{x}_1, \overline{x}_1] \times \ldots \times [\underline{x}_n, \overline{x}_n]$ a box in $\mathbb{R}^n$. The *box symmetry* function S is defined as $S(\mathcal{B}) = \{s(\mathbf{x}) \;\; s.t. \;\; \mathbf{x} \in \mathcal{B}\} = [\underline{x}_{\theta(1)}, \overline{x}_{\theta(1)}] \times \ldots \times [\underline{x}_{\theta(n)}, \overline{x}_{\theta(n)}] = [\underline{x}_2, \overline{x}_2] \times \ldots \times [\underline{x}_n, \overline{x}_n] \times [\underline{x}_1, \overline{x}_1]$. The box symmetry function has also an associated constraint permutation σ, which is the same associated to s. S^i will denote S composed i times. We say, then, that $S^i(\mathcal{B})$ and $S^j(\mathcal{B})$ are symmetric boxes, $0 \leqslant i, j < n$, $i \neq j$.

Box symmetry is an equivalence relation defining symmetry equivalence classes. Let $R(\mathcal{B})$ be the set of *different* boxes in the symmetry class of $\mathcal{B}$, $R(\mathcal{B}) = \{S^i(\mathcal{B}), i \in \{0, \ldots, n-1\}\}$. For instance, for box $\mathcal{B}' = [0, 4] \times [2, 5] \times [2, 5] \times [0, 4] \times [2, 5] \times [2, 5]$, $R(\mathcal{B}')$ is composed of $S0(\mathcal{B}') = \mathcal{B}'$, $S1(\mathcal{B}') = [2, 5] \times [2, 5] \times [0, 4] \times [2, 5] \times [2, 5] \times [0, 4]$ and $S2(\mathcal{B}') = [2, 5] \times [0, 4] \times [2, 5] \times [2, 5] \times [0, 4] \times [2, 5]$. Note that $S3(\mathcal{B}')$ is again $\mathcal{B}'$ itself and that subsequent applications of box symmetry would repeat the same sequence of boxes. We define the *period* $P(\mathcal{B})$ of a

[2] This set $\{s(\mathbf{x}) \;\; s.t. \;\; \mathbf{x} \in \mathcal{B}\}$ is also a box if $s(\mathbf{x}) = (s_1(\mathbf{x}), \ldots, s_n(\mathbf{x})) = (g_1(x_{\phi(1)}), \ldots, g_n(x_{\phi(n)}))$, where s_i is the i-th component of s, ϕ is an arbitrary permutation and $g_i : \mathbb{R} \to \mathbb{R}$ is any function such that if I is an interval of $\mathbb{R}$ then $\{g_i(x) \; s.t. \; x \in I\}$ is also an interval of $\mathbb{R}$.

box $\mathcal{B}$ as $P(\mathcal{B}) = |R(\mathcal{B})|$. It is easily shown that $R(\mathcal{B}) = \{S^i(\mathcal{B}), i \in \{0, \ldots, P(\mathcal{B}) - 1\}\}$. For example, for box $\mathcal{B}'$, $R(\mathcal{B}') = \{S0(\mathcal{B}'), S1(\mathcal{B}'), S2(\mathcal{B}')\}$ and $P(\mathcal{B}') = 3$.

As in the case of point symmetry, box symmetry has implications for the continuous CSP. If there is no solution inside a box $\mathcal{B}$, there is no solution inside any of its symmetric boxes either. A box $\mathcal{B}_f \subseteq \mathcal{B}$ is a solution iff $S^i(\mathcal{B}_f) \subseteq S^i(\mathcal{B})$ is a solution box for any $i \in \{1 \ldots P(\mathcal{B}) - 1\}$.

In the following sections we will show that the implications of box symmetry for a continuous CSP can be exploited to save much computing time in a meta-algorithm that uses a CCS as a tool without interfering with it.

3.1 Box symmetry classes obtained by bisecting a n-cube

The algorithm we will propose to exploit box symmetry makes much use of the symmetry classes formed by bisecting a n-dimensional cube I^n (i.e., of period 1) in all dimensions at the same time and at the same point, resulting in 2^n boxes. We will denote L and H the two subintervals into which the original range I is divided. For example, for $n = 2$, we have the following set of boxes $\{L \times L, L \times H, H \times L, H \times H\}$ whose periods are 1, 2, 2 and 1, respectively. And their symmetry classes are: $\{L \times L\}$, $\{L \times H, H \times L\}$, and $\{H \times H\}$. Representing the two intervals L and H as 0 and 1, respectively, and dropping the $\times$ symbol, the sub-boxes can be coded as binary numbers. Let $\mathbb{SR}_n$ be the set of representatives, formed by choosing the smallest box in binary order from each class. For example, $\mathbb{SR}_2 = \{00, 01, 11\}$. Note that the cube I^n to be partitioned can be thought of as the the set of binary numbers of length n, and that $\mathbb{SR}_n$ is nothing more than a subset whose elements are different under circular shift.

The algorithm for exploiting symmetries and the way it uses $\mathbb{SR}_n$ are explained in the next section. Afterwards, in Sections 6 and 7, we study how many components $\mathbb{SR}_n$ has, how they are distributed and, more importantly, how can they be generated.

4 Algorithm to exploit box symmetry

The symmetry exploitation algorithm we propose uses the CCS as an external routine. The internals of the CCS must not be modified or known.

The idea is to first divide the initial box into a number of symmetry classes. Next, one needs to process only a representative of each class with the CCS. At the end, by applying box symmetries to the solution boxes obtained in this way, one would get all the solutions lying in the space covered by the whole classes, i.e., the initial box. The advantage of this procedure is that the CCS would have to process only a fraction of the initial box. Assuming that the initial box is a n-cube covering the same interval $[x_l, x_h]$ in all dimensions, we can directly apply the classes associated to $\mathbb{SR}_n$. A procedure to exploit single-cycle symmetries in this way is presented in Algorithm 1.

Since $\mathbb{SR}_n$ is a set of codes –not real boxes– we need a translation of the codes into boxes for the given initial box. The operator GenerateSubBox($\mathbf{b}, x_l, x_h, x^*$) returns the box $\mathcal{V} = V_1 \times \cdots \times V_n$ corresponding to code $\mathbf{b} = b_1 \ldots b_n$ when $[x_l, x_h]$ is the range of the initial box in all dimensions and x^* is the point in which this interval is bisected:

$$V_i = \begin{cases} [x_l, x^*] & \text{if } b_i = 0, \\ [x^*, x_h] & \text{if } b_i = 1. \end{cases} \tag{1}$$

The point x^* calculated by SelectBisectionPoint(x_l, x_h) can be any such that $x_l < x^* < x_h$, but a reasonable one is $(x_l + x_h)/2$. The iterations over line 4 generate a set of representative boxes such that, together with their symmetries, cover the initial n-cube.

ProcessRepresentative$(\mathcal{B})$ returns all the solution boxes associated to $\mathcal{B}$, that is, the solutions inside $R(\mathcal{B})$, or still in other words, the solutions inside $\mathcal{B}$ and inside its symmetric boxes. Since the number of symmetries of $\mathcal{B}$ is $P(\mathcal{B})$, the benefits of exploiting the symmetries of a class representative is proportional to its period.

Algorithm 1: CSym algorithm.

Input: A n-cube, $[x_l, x_h] \times \cdots \times [x_l, x_h]$.
A single-cycle box symmetry, S.
A Continuous Constraint Solver, CCS.
Output: A set of boxes covering all solutions.

```
1 SolutionBoxSet ← EmptySet
2 x* ← SelectBisectionPoint(x_l, x_h)
3 foreach b ∈ 𝕀ℝ_n do
4 |   B ← GenerateSubBox(b, x_l, x_h, x*)
5 |   SolutionBoxSet ← SolutionBoxSet ∪ ProcessRepresentative(B)
6 return SolutionBoxSet
```

Algorithm 2: The ProcessRepresentative function.

Input: A box, $\mathcal{B}$.
A single-cycle box symmetry, S.
A Continuous Constraint Solver, CCS.
Output: The set of solution boxes contained in $\mathcal{B}$ and its symmetric boxes.

```
1 SolSet ← CCS(B)
2 TotalSolSet ← SolSet
3 for i=1: P(B) − 1 do
4 |   TotalSolSet ← TotalSolSet ∪ ApplySymmetry(SolSet, S^i)
5 return TotalSolSet
```

The correctness of the algorithm is easy to check. The set of boxes in which it searches explicitly or implicitly (by means of symmetry) for solutions is $\mathcal{U} = \{R(\mathcal{B})\ s.t.\ \mathcal{B}$ is a representative$\}$. In fact, $\mathcal{U}$ is the set of boxes formed by bisecting the initial box in all dimensions at the same time and at the same point. $\mathcal{U}$ covers the whole initial box, and thus, the algorithm finds *all* the solutions of the problem. Moreover, it finds each solution box only *once*, because the boxes in $\mathcal{U}$ do not have any volume in common (they share at most a "wall").

5 An illustrative example

Molecules can be modeled as mechanical chains by making some reasonable approximations. If two atoms are joined by a chemical bond, one can assume that there is a rigid link between them. Thus, the first approximation is that bond lengths are constant. The second one is that the angles between two consecutive bonds are also constant. In other words, the distances between the atoms in any subchain of three atoms are assumed to be constant. All configurations of the atoms of the molecule that satisfy these distance constraints, sometimes denoted *rigid-geometry*

hypothesis, are valid conformations of the molecule in a kinematic sense. The constraints induced by the rigid-geometry hypothesis are particularly strong when the molecule topology forms loops, as in cycloalkanes. The problem of finding all valid conformations of a molecule can be formulated as a distance-geometry [2] problem in which some distances between points (atoms) are fixed and known, and one must find the set of values of unknown (variable) distances that are compatible with the embedding of the points in $\mathbb{R}3$. The unknown distances can be found by solving a set of constraints consisting of equalities or inequalities of determinants formed with subsets of the fixed and variable distances [2].

The problem can be solved using a CCS [10, 9]. Figure 1 displays the known and unknown distances of the cycloheptane, a molecule basically composed of a ring of seven carbon atoms. The distance between two consecutive atoms of the ring is constant and equal everywhere. The distance between two atoms connected to a same atom is also known and constant no matter the atoms. The problem has several symmetries. One of them is $s(d_1, \ldots, d_7) = (d_{\theta(1)}, d_{\theta(2)}, \ldots, d_{\theta(7)}) = (d_2, d_3 \ldots, d_7, d_1)$. When this symmetry is exploited with the CSym algorithm the problem is solved in 4.64 minutes, which compares very favorably with the 31.6 minutes spent when using the algorithm in [9] alone. Thus, a reduction by a factor close to $n = 7$ (i.e., the length of the symmetry cycle) in computing time is obtained, which suggests that the handling of box symmetries doesn't introduce a significant overhead. Figure 2 shows a projection into d_1 d_2 and d_3 of the solutions obtained using CSym.

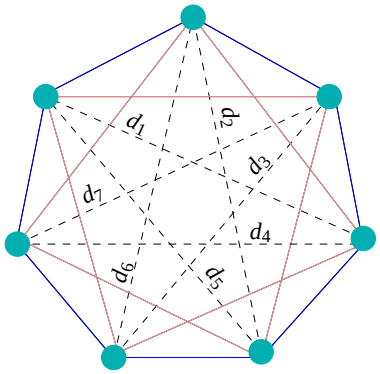


Figure 1: Cycloheptane. Disks represent carbon atoms. Constant and variable distances between atoms are represented with continuous and dashed lines, respectively.

6 Analysis of $\mathbb{SR}_n$: Counting the number of classes

Let us define some quantities of interest:
-$\mathcal{N}_n$: Number of elements of $\mathbb{SR}_n$.
-$\mathcal{FP}_n$: Number of elements of $\mathbb{SR}_n$ that correspond to full-period boxes, i.e., boxes of period n.
-$\mathcal{N}_{nm}$: Number of elements of $\mathbb{SR}_n$ having m 1's.
-$\mathcal{FP}_{nm}$: Number of elements of $\mathbb{SR}_n$ that correspond to full-period boxes having m 1's.

Polya's theorem [8] could be used to determine some of these quantities for a given n by building a possibly huge polynomial and elucidating some of its coefficients. We present a simpler way of calculating them and, at the same time, make the reader familiar with the concepts that will be used in our algorithm to generate $\mathbb{SR}_n$.

We begin by looking for the expression of $\mathcal{FP}_n$. When any number of 1's is allowed, the total number of binary numbers is 2^n. The only periods that can exist in these binary numbers are

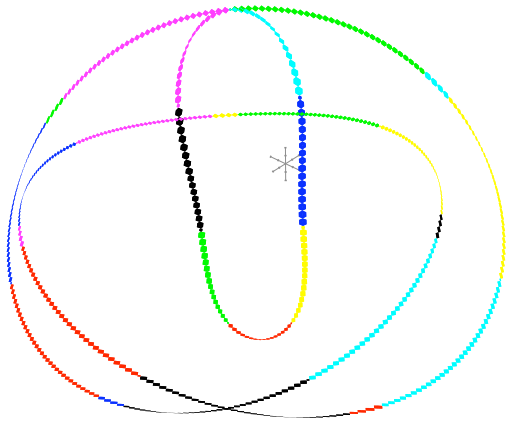


Figure 2: Three-dimensional projection of the cycloheptane solutions. The lightest (yellow) boxes are the solutions found inside the representatives using the CCS (line 1 in Algorithm 2). The other colored boxes are the solutions obtained by applying symmetries to the yellow boxes (line 4 in Algorithm 2).

divisors of n. Thus, the following equation holds:

$$\sum_{p \in div(n)} p \; \mathcal{FP}_p = 2^n. \tag{2}$$

Segregating $p = n$,

$$n \; \mathcal{FP}_n + \sum_{p \in div(n),\; p<n} p \; \mathcal{FP}_p = 2^n, \tag{3}$$

and solving for $\mathcal{FP}_n$:

$$\mathcal{FP}_n = \frac{2^n}{n} - \sum_{p \in div(n),\; p<n} \frac{p}{n} \; \mathcal{FP}_p. \tag{4}$$

This recurrence has a simple baseline condition: $\mathcal{FP}_1 = 2$.

Then, $\mathcal{N}_n$ follows easily from

$$\mathcal{N}_n = \sum_{p \in div(n)} \mathcal{FP}_p. \tag{5}$$

Segregating $p = n$, a more efficient formula is obtained:

$$\mathcal{N}_n = \frac{2^n}{n} + \sum_{p \in div(n),\; p<n} \frac{n-p}{n} \; \mathcal{FP}_p. \tag{6}$$

This formula is valid for $n > 1$. The remaining case is $\mathcal{N}_1 = 2$.

We will use similar techniques to obtain $\mathcal{FP}_{nm}$ and $\mathcal{N}_{nm}$. There are $\binom{n}{m}$ binary numbers having m 1's and $n-m$ 0's. Some of these binary numbers are circular shifts of others (like 011010 and 110100). The number of shifted versions of a same binary number is the period of the box being represented by the binary number. For example, 1010, of period 2, has only another shifted version, 0101. A binary number representing a box of period p can be seen as a concatenation of n/p numbers of length $\frac{n}{n/p} = p$ and period p. This means that these "concatenated" numbers are full-period, and they have $\frac{m}{n/p}$ 1's. Thus, the number of binary numbers of period p when shifted numbers are counted as the same (i.e., the number of classes of period p) is $\mathcal{FP}_{\frac{n}{n/p}\frac{m}{n/p}}$. Only common divisors of n and m, which we denote $div(n,m)$, can be periods. Since there are p shifted versions of each binary number having period p, we can write

$$\sum_{p \in div(n,m)} p\ \mathcal{FP}_{\frac{n}{n/p}\frac{m}{n/p}} = \binom{n}{m}. \tag{7}$$

With a change of variable $f = n/p$ we get

$$\sum_{f \in div(n,m)} \frac{n}{f}\ \mathcal{FP}_{\frac{n}{f}\frac{m}{f}} = \binom{n}{m}. \tag{8}$$

Note that the index of the summation goes through the same values as before. We can segregate the case $f = 1$ from the summand,

$$n\ \mathcal{FP}_{nm} + \sum_{f \in div(n,m),\ f>1} \frac{n}{f}\ \mathcal{FP}_{\frac{n}{f}\frac{m}{f}} = \binom{n}{m}, \tag{9}$$

and, finally, we obtain

$$\mathcal{FP}_{nm} = \frac{\binom{n}{m}}{n} - \sum_{f \in div(n,m),\ f>1} \frac{\mathcal{FP}_{\frac{n}{f}\frac{m}{f}}}{f}. \tag{10}$$

This is a recurrence relation from which $\mathcal{FP}_{nm}$ can be computed using the following baseline conditions:

$$\mathcal{FP}_{nn}, \mathcal{FP}_{n0} = \begin{cases} 0 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases} \tag{11}$$

$\mathcal{N}_{nm}$ is obtained adding up the number of classes of each period:

$$\mathcal{N}_{nm} = \sum_{f \in div(n,m)} \mathcal{FP}_{\frac{n}{f}\frac{m}{f}}. \tag{12}$$

Segregating again $f = 1$, a more efficient formula is obtained:

$$\mathcal{N}_{nm} = \binom{n}{m} + \sum_{f \in div(n,m),\ f>1} (1 - \frac{n}{f})\mathcal{FP}_{\frac{n}{f}\frac{m}{f}}, \tag{13}$$

then carrying out the change of variable $p = n/f$:

$$\mathcal{N}_{nm} = \binom{n}{m} + \sum_{p \in div(n,m),\ p<n} (1-p)\mathcal{FP}_{p\frac{mp}{n}}, \tag{14}$$

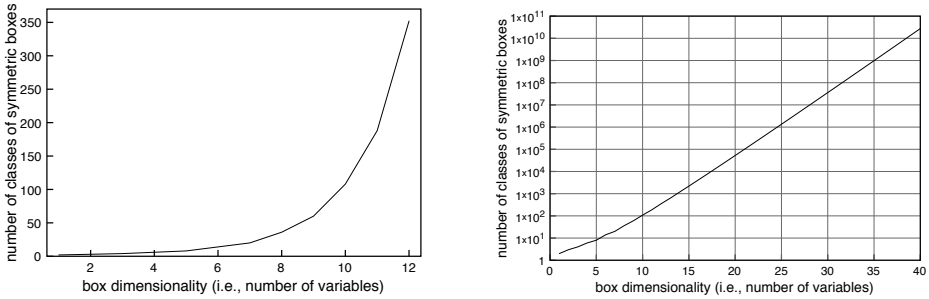
Note the change in the summation range. This equation is valid whenever $m > 0$ and $m < n$. Otherwise, $\mathcal{N}_{nm} = 1$.

It is possible to extend the concept of $\mathcal{FP}_n$ (and $\mathcal{FP}_{nm}$) to reflect the number of members of $\mathbb{SR}_n$ having period p (and m 1's), which we denote $\mathcal{N}_n^p$ ($\mathcal{N}_{nm}^p$):

$$\mathcal{N}_n^p = \begin{cases} 0 & \text{if } p \notin div(n) \\ \mathcal{FP}_p & \text{otherwise} \end{cases} \tag{15}$$

$$\mathcal{N}_{nm}^p = \begin{cases} 0 & \text{if } p \notin div(n,m) \\ \mathcal{FP}_{p,\frac{mp}{n}} & \text{otherwise} \end{cases} \tag{16}$$

Figure 3(a) displays the number of classes ($\mathcal{N}_n$) as a function of n. The curve indicates an exponential-like behavior. This is confirmed in Figure 3(b) using a larger logarithmic scale, in which the curve appears almost perfectly linear. Figure 4 is an example of the distribution of classes by period for $n = 12$. Figure 5 shows the percentage of full-period classes in $\mathbb{SR}_n$ ($100\ \mathcal{N}_n^n/\mathcal{N}_n$). One can see that the percentage of classes with period different from n is significant for low n, but approaches quickly 0 as n grows. Finally, Figures 6(a) and 6(b) display the distribution of the classes in $\mathbb{SR}_n$ by number of 1's for $n = 12$ and $n = 100$, respectively. The majority of the classes concentrates in an interval in the middle of the graphic, around $n/2$. This interval becomes relatively smaller when n grows.



(a) (b)

Figure 3: Number of elements of $\mathbb{SR}_n$ as a function of n.

7 Generating $\mathbb{SR}_n$, the classes of symmetric boxes

The naive procedure to obtain $\mathbb{SR}_n$ would initially generate all boxes originated by bisecting a n-dimensional cube at the same point in all dimensions at the same time. Then, one should check each of the boxes in this set to detect whether it is a circular shift of some of the others. The complete process of generating $\mathbb{SR}_n$ in this way involves a huge number of operations even for rather small dimensions. Although the $\mathbb{SR}_n$ for a few n's could be pre-computed and stored in a

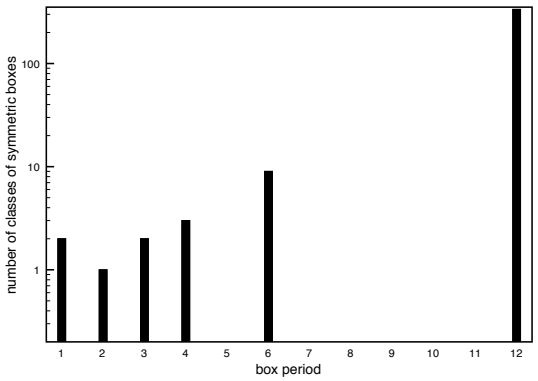


Figure 4: Number of elements of $\mathbb{SR}_{12}$ distributed by period.

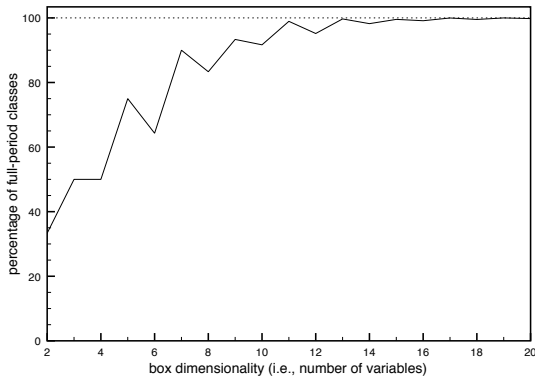
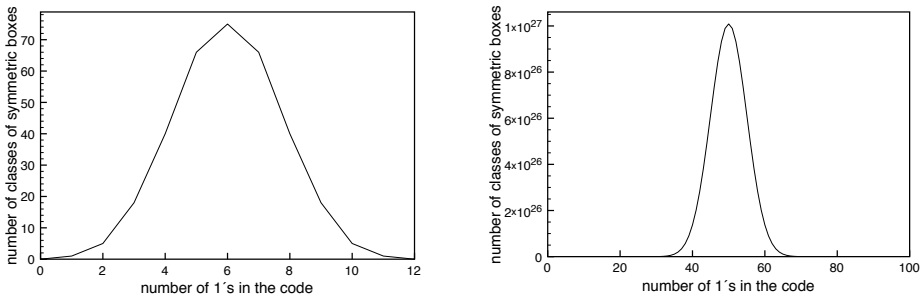


Figure 5: Percentage of full-period elements in $\mathbb{SR}_n$ as a function of n.



(a) (b)

Figure 6: Number of elements of $\mathbb{SR}_n$ distributed by number of 1's. (a) n =12. (b) n=100.

database, we suggest here an algorithm capable of calculating $\mathbb{SR}_n$ on the fly without significant computational overhead.

As made for counting, we distinguish different subsets of $\mathbb{SR}_n$ on the basis of the number of 1's and the period:

-$\mathbb{SR}_{nm}$: Subset of the elements of $\mathbb{SR}_n$ having m 1's.

-$\mathbb{SR}^p_{nm}$: Subset of the elements of $\mathbb{SR}_n$ having m 1's and period p.

–$\mathbb{SR}^p_n$: Subset of the elements of $\mathbb{SR}_n$ having period p.

From a global point of view, the generation of $\mathbb{SR}_n$ is carried out as follows. First, $\mathbb{SR}_{n0}$ is generated, which is constituted always by a unique member. Afterwards, all $\mathbb{SR}_{nm}$ for $m = 1 \ldots n$ are generated. The generation of $\mathbb{SR}_{nm}$ is divided in each of the $\mathbb{SR}^p_{nm}$, $p \in div(n, m)$, that compose it. Since the efficient algorithm we describe below generates $\mathbb{SR}^n_{nm}$, in order to obtain $\mathbb{SR}^p_{nm}$ we generate $\mathbb{SR}^{\frac{n}{f}}_{\frac{n}{f}\frac{m}{f}} = \mathbb{SR}^p_{p\frac{mp}{n}}$, where $f = n/p$, and concatenate their elements f times.

We use a compact coding of the binary numbers representing the boxes consisting in ordered lists or chains of numbers. The first number of the code is the number of 0's appearing before the first 1 in the binary number. The i-th number of the code for $i > 1$ is the number of 0's between the $(i-1)$-th and the i-th 1's of the binary number. For example, the number 0100010111 is codified as 13100. The length of this numerical codification is the number of 1's of the codified binary number, which has been denoted by m.

There are binary numbers that cannot be codified in this way, because their last digit is 0. But, except for the all zero's case, there is always an element of its class that can be codified correctly (for example 0011 is an element of the class of 0110). As our objective is to have only a representative of each class, this is rather an advantage, because half of the boxes are already eliminated from the very beginning. The all zero's box, $\mathbb{SR}_{n0}$, is common to every n, and will be generated separately, as already mentioned.

The codification allows to determine if a box is full-period in the same way as in the binary representation: the box has period n iff after a number of circular shifts lower than the length of the numerical chain the result is never equal to the original. For instance, the example above is full-period, but 22, corresponding to 001001, is not. The only difference is that, in the new representation, at most m shifts must be compared.

The code of a box can be seen as a number of base $n - m$. In a full-period box, the m circular shifts of the code are different numbers, and can be arranged in strictly increasing numerical order. We will take as representative box of a class the largest element of the class when expressed as a code. For example, the class of 130 has two other elements that can be represented by our coding, 013 and 301, the latter being the chosen representative of the class.

Note that a box belonging to $\mathbb{SR}_{nm}$ has $n - m$ 0's or, equivalently, the sum of the components of the code is $n - m$.

The output of the algorithm are all codes of length m, whose sum of components is $n - m$, and which are both representatives of a class and full-period. Codes of length m whose components sum up a desired number are rather easy to generate systematically. The representativeness and full-period conditions are more difficult to guarantee efficiently. We can handle them by exploiting the following properties of our codes:

Property 1 *If a sub-chain of the code beginning at position $i > 1$ and ending at the last position*

m (thus of length m-i+1) is greater or equal in numerical terms than the sub-chain of the same length beginning at the first position, then either the code is not a representative or the code is not full-period. In the opposite case (i.e., when the "head" sub-chain is greater), we say that the code is i-compatible or compatible for position i.

Therefore, a code whose position is not i-compatible for some $1 < i \leqslant m$ is not a valid output for the algorithm.

Property 2 *If a code is i-compatible for all i s.t. $1 < i \leqslant m$, it is a class representative and it is full-period.*

Thus, instead of comparing chains of length m, we can determine the code validity comparing shorter sub-chains. A third property help us to devise a still faster and simpler algorithm:

Property 3 *If a code is i-compatible and the sub-chain from position i to $i + l$ is equal to the sub-chain from position 1 to $1 + l$ then the code is also compatible for positions $i + 1$ through $i + l$.*

This property is interesting because it permits checking the validity of the code by travelling along it at most once, as shown in Algorithm 3. The trick is that when the decision of i-compatibility is being delayed because position i and the following numbers are the same as those at the beginning of the string, if it finally resolves positively, the compatibility for the intermediate numbers is also guaranteed. Hence, i-compatibility is either resolved with a simple comparison or it requires l comparisons. In the latter case, either the compatibility of l positions is also resolved (if the outcome is positive) or compatibility of intermediate positions doesn't matter (because the outcome is negative and, thus, the code can be labelled non valid without further checks). A *ctrol* variable is in charge of maintaining the last index of the "head" sub-chain that is being compared in the current compatibility check. When examining the compatibility of the current position i, if its value is lower than that of the *ctrol* position, the code is for sure i-compatible and therefore we must only worry about $(i+1)$-compatibility by back-warding *ctrol* to the first position. If the value of the *ctrol* position is equal to that of the current position i, the compatibility of position i is still to be ascertained, and we continue advancing the current and the *ctrol* positions until the equality disappears. In other words, the only condition that must be fulfilled for non rejecting as invalid a code at position i is that

$$A[i] \leqslant A[ctrol], \tag{17}$$

a condition that is transformed into $A[i] < A[ctrol]$ when $i = m$ to resolve the last of the pending compatibility checks. As an aside, note that our codes are more general than the raw binary numbers, and that representativeness and full-periodness are defined in the same way for both. Therefore, the three properties and the CodeValidity algorithm apply also to the raw binary numbers.

Our main procedure to obtain $\mathbb{SR}^n_{nm}$ is the recursive program presented in Algorithm 4. ClassGen$(n-m, 1, 1, m, A)$, where A is an array of length m, must be called to obtain $\mathbb{SR}^n_{nm}$, for $n > 1$, $m > 0$. Each call to the procedure writes a single component of the code at the position of A indicated by the parameter *pos*, which is subsequently incremented. The first parameter, *sum*, is the sum of the components of the code that remain to be written. The number written in the current position is subtracted from the *sum* parameter in the next recursive call. At the end of the code (in the case of $pos = m$), only *sum* is allowed to be written, which guarantees that the code, if accepted, will have sum of components equal to $n - m$. The acceptance condition for $pos = m$ (line 7) is the value being strictly minor than $A[ctrol]$, just as in CodeValidity at the end of the code.

Algorithm 3 (presented only for clarity purposes) is not used. Instead, full-period and representativeness conditions are enforced within Algorithm 4 itself by the range of values allowed to be written at the current position *pos*. This range is limited by *LowerLimit* and *UpperLimit*. In the general case ($pos \neq 1$, $pos \neq m$) described in lines 7-9, these limits reflect the same condition (17) expressed for the CodeValidity algorithm, that is, the value of A at the current position must be less or equal than that at the *ctrol* position. Moreover, in this case, we can neither allow anything greater than the sum of the numbers to be writen, *sum*. The maintenance of the *ctrol* variable is also similar to that within the CodeValidity algorithm: if we write in *pos* something strictly minor than $A[ctrol]$, *ctrol* is back-warded to the first position. Otherwise, *ctrol* is incremented by 1 for the next recursive call to write $pos + 1$. In the case of $pos = 1$ there is no upper limit enforced by the representativeness and full-periodness conditions. Consequently, *UpperLimit* is equal to *sum*. *LowerLimit* can also be determined accurately since, for a value minor than $\lceil sum/m \rceil$, there is no way to distribute what remains of *sum* among the other positions of the code without putting a value greater than the initial one, which would make any such code non-representative.

The output of the algorithm is a list of valid codes in decreasing numerical order. For instance, the output obtained when requesting $\mathbb{SR}9_{93}$ with ClassGen$(6, 1, 1, 3, A)$ is: {600, 510, 501, 420, 411, 402, 330, 321, 312}. In this example, the only case in which the recursion arrives to $pos = m$ without returning a valid code is the frustrated code 222, whose last number is not written because the code is not full-period.

Algorithm 3: CodeValidity algorithm.

Input: A code of length m expressed as an array, A.

Output: A boolean value indicating whether the code is valid, i.e., whether it is full-period and a class representative.

```
1  i ← 2
2  ctrol ← 1
3  ValidCode ← True
4  while ValidCode & i < m do
5  |   if A[i] > A[ctrol] then ValidCode ← False
6  |   else if A[i] < A[ctrol] then ctrol ← 1
7  |        else ctrol ← ctrol + 1;                    /* A[i] = A[ctrol] */
8  |   i ← i + 1
9  if A[m] ≥ A[ctrol] then ValidCode ← False
10 return ValidCode
```

8 Conclusions

We have approached the problem of exploiting symmetries in continuous constraint satisfaction problems using continuous constraint solvers. Our approach is general and could be used also with other box-oriented algorithms, such as Branch-and-Bound for nonlinear optimization. The particular symmetries we have tackled are single-cycle permutations of the problem variables.

The suggested strategy is to bisect the domain, the n-cube initial box, simultaneously in all dimensions at the same point. This forms a set of boxes that can be grouped in box symmetry classes. A representative of each class is selected to be processed by the CCS and all the

Algorithm 4: ClassGen algorithm.

Input: The sum of the numbers that remain to be written on the right (from position *pos* to m), *sum*.
The index of the next position to be written, *pos*.
The index of the current control element, whose value cannot be surpassed in the next position, *ctrol*.
The length of the code, m.
Array where class codes are being generated, A.

Output: A set of codes representing classes, $\mathbb{SR}$.

```
1  SR ← EmptySet
2  if pos = m then
3      if sum < A[ctrol] then                    /* otherwise, SR will remain EmptySet */
4          A[m] ← sum
5          SR ← {A};
6  else
7      if pos ≠ 1 then
8          LowerLimit = 0
9          UpperLimit ← Minimum(A[ctrol], sum)
10     else
11         LowerLimit = ⌈sum/m⌉
12         UpperLimit ← sum
13     for i = UpperLimit to LowerLimit do
14         A[pos] ← i
15         if i = A[ctrol] then                          /* i = A[ctrol] = UpperLimit */
16             SR ← SR ⋃ ClassGen(sum − i, pos + 1, ctrol + 1, m, A)
17         else SR ← SR ⋃ ClassGen(sum − i, pos + 1, 1, m, A);          /* i < A[ctrol] */
18 return SR
```

symmetries of the representative are applied to the resulting solutions.

In this way, the solutions within the whole initial domain are found, while having processed only a fraction of it –the set of representatives– with the CCS. The time savings obtained by processing a representative and applying its symmetries to the solutions tend to be proportional to the number of symmetric boxes of the representative. Therefore, symmetry exploitation is complete for full-period representatives, since they have the maximum number of symmetric boxes.

We have also studied the automatic generation of the classes resulting from bisecting a n-cube and analyzed their numerical properties. The algorithm for generating the classes is very powerful, eliminating the convenience of any pre-calculated table. The numerical analysis of the classes revealed that the average number of symmetries of the class representatives tends quickly to n as the number of variables, n, grows. These are good news, since n is the maximum number of symmetries attainable with single-cycle symmetries of n variables, leading to time reductions by a factor close to n. Nevertheless, for small n there is still a significant fraction of the representatives not having the maximum number of symmetries. Another weakness of the proposed strategy is the exponential growth in the number of classes as a function of n.

The problems with small and large n should be tackled with a more refined subdivision of the

initial domain in box symmetry classes, which is left for near future work. We are also currently approaching the extension of this work to deal with permutations of the problem variables composed of several cycles.

References

[1] Benhamou F., Goulard F.: Universally Quantified Interval Constraints. Proc. 6th CP (2000) 67-82

[2] Blumenthal, L.: Theory and aplications of distance geometry. Oxford University Press, 1953.

[3] Cohen D., Jeavons P., Jefferson Ch., Petrie K.E., Smith B.M.: Symmetry Definitions for Constraint Satisfaction Problems. Constraints **11(2-3)** (2006) 115-137

[4] Flener P., Frisch A.M., Hnich B., Kiziltan Z., Miguel I., Pearson J., Walsh, T.: Breaking Row and Column Symmetries in Matrix Models. Proc. 8th CP (2002) 462-476

[5] Gent I.P., Harvey W., Kelsey T.: Groups and Constraints: Symmetry Breaking During Search. Proc. 8th CP (2002) 415-430

[6] Jermann C., Neveu B., Trombettoni G.: Inter-Block Backtracking: Exploiting the Structure in Continuous CSPs. 2nd International Workshop on Global Constrained Optimization and Constraint Satisfaction (Cocos'03)

[7] Meseguer P., Torras C.: Exploiting symmetries within constraint satisfaction search. Artificial Intelligence **129** (2001) 133-163

[8] Polya, G. and Read, R.C.: Combinatorial enumeration of groups, graphs and chemical compounds. Springer-Verlag, New York, 1987

[9] Porta, J.M., Ros Ll., Thomas F., Corcho F., Canto J. and Perez J.J.: Complete maps of molecular loop conformational spaces. Journal of Computational Chemistry **28 (13)** (2007) 2170-2189

[10] Porta J.M., Ros Ll., Thomas F., Torras C.: A Branch-and Prune Solver for Distance Constraints. IEEE Trans. on Robotics **21(2)** (2005) 176-187

[11] Puget J-F.: Symmetry Breaking Revisited. Constraints **10(1)** (2005) 23-46

[12] Ruiz de Angulo V., Torras C.: Exploiting single-cycle symmetries in branch-and-prune algorithms. Proc. 13th CP (2007) 864-871

[13] Sam-Haroud D., Faltings B.: Consistency Techniques for Continuous Constraints. Constraints **1(1-2)** (1996) 85-118

[14] Vu X-H., Silaghi M., Sam-Haroud D., Faltings B.: Branch-and-Prune Search Strategies for Numerical Constraint Solving. Swiss Federal Institute of Technology (EPFL) LIA-REPORT **7** (2006)

Acknowledgements

This work was supported by the I+D pro ject DPI 2004-07358 of the Spanish Ministry of Education.

IRI reports

This report is in the series of IRI technical reports.
All IRI technical reports are available for download at the IRI website
`http://www-iri.upc.edu`.